“LET’S COOK”

Architecture

Ce document dresse un constat général sur l’architecture du projet « Let’s cook » réalisé par le groupe n°5 dans le cadre du cours de génie logiciel.

François REMY, Nicolas BERNIER, Sanawar SYED AZOR ALI,
Thibaut NICODEME, Ekaterina ASFANDYAROVA

Table des matières

Contents

Your diagram definitely needs more arrows.

Someone on Twitter

Architecture générale

Organisation des packages

L'implémentation est séparée en trois packages principaux:

- **com.letscook.*** : packages pour l'application et ses activités ainsi que les modèles de la vue.
- **com.letscook.models.*** : packages incluant notre modèle de données et leurs interfaces
- **db.*** : packages permettant la liaison avec la base de données.

En résumé, chaque package fonctionne de la façon suivante :

COM.LETSCOOK.* : Les Activités sont les classes invoquées par les fichiers XML de la vue. Elles se créent chacune une ou plusieurs classe du modèle de vues, permettant de les relier au modèle (et donc à la base de données); La responsabilité des activités est de charger la vue et de s'occuper de faire la liaison avec Android.

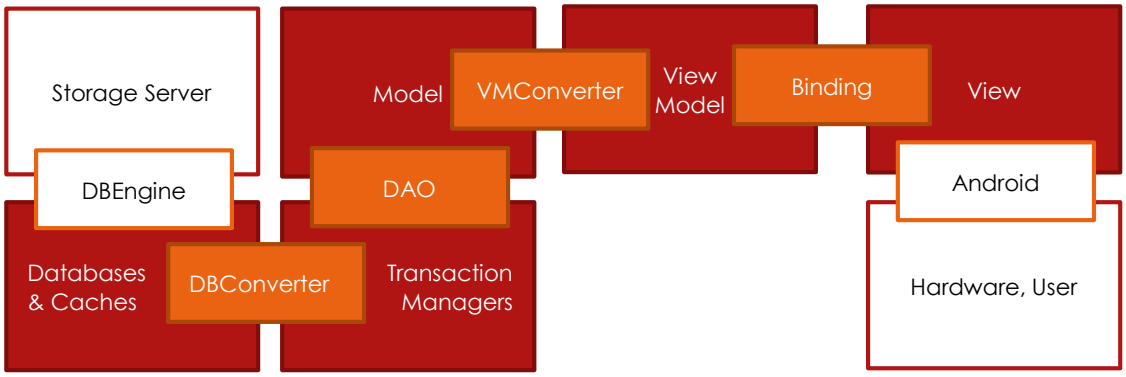
COM.LETSCOOK.UTIL.* : Le package contient des helpers et des classes primitives implémentant des fonctions utiles mais ne faisant pas référence à une classe particulière. Par exemple, la classe « FileHelpers » permet de simplifier l'utilisation de fichiers de sérialisation, et la classe « IntentBroker » permet de gérer les contrats entre activités.

COM.LETSCOOK.MODELS.* : Il s'agit du modèle, accédant à la base de données pour créer les données utilisées par l'application. Chaque objet utilise plusieurs interfaces permettant d'y accéder de façon différentes (en lecture ou en écriture, et en tant qu'objet général ou spécifique (supermarché, ingrédient, recette, etc.)). Le package models.nodb permet de créer des objets temporaires, qui ne s'insèrent pas automatiquement dans la base de données; hormis cela, il est identique au package models.

DB.* : Ce package permet l'interaction avec la base de données. Le DataAccessObject permet d'aller chercher un objet spécifique ou une liste d'objet basée sur une requête exercée sur la base de données.

Les sous-packages ont les fonctions suivantes:

- db.binding définit des classes utilisées comme attribut par le modèle, et qui se mettent automatiquement à jour depuis la base de données en s'auto-observant.
- db.remote gère l'accès à une base de données distante
- db.local gère l'accès à une base de données locale
- db.mixed gère l'accès cache/maitre à deux bases de données



Patrons de conception

Patron MVVM

Pour notre architecture globale, nous avons utilisé le patron structurel MVVM :

- **Model** Représente les objets venant des données
- **View** Représente les interfaces graphiques (Activity + xml)
- **VModel** Représentes les données de la GUI, ce qui est affiché ainsi que les interactions

Pour un objet « Market » de notre modèle, on construit par exemple :

- **Model** models.Market
- **View** app.views.MarketViewer (+layout)
  app.views.MarketChooser (+layout)
- **VModel** app.views.models.Market_VModel
  app.views.models.MarketList_VModel

Le modèle d'une vue est bien entendu composée de modèles de données, dont il s'occupe parfois de veiller à la conversion (exemple : *DoubleToCharSequenceObservable*).

Pour implémenter une telle architecture, nous avons utilisé une libraire : Android Binding, permettant de réaliser ce patron aisément.

Patron Observer

Le patron MVVM nécessite l'utilisation de classes Observable/*Observers*, pour permettre aux différents éléments de l'architecture de se mettre à jour automatiquement quand l'un d'eux est mis à jour. Ce patron est implémenté sous la forme de la librairie Android Binding, décrite ci-dessous.

L'Android Binding s'utilise en rajoutant directement dans les layouts xml des attributs « binding ».

```
<LinearLayout ... binding:visibility="hasNoResult" binding:onClick="cancel">
```

Ils permettent notamment de faire office de contrôleur grâce aux événements … Ceux-ci permettent d'appeler directement des commandes. Ces commandes sont à implémenter dans le modèle de la vue. Cela permet d'éviter d'écrire tout le code d'interaction dans une activité et de se concentrer sur l'action en elle-même au niveau où elle a lieu.
Ces attributs permettent également de lier la valeur courante d'une zone de texte par exemple à un observable qui, s'il est modifié dans le modèle de la vue sera automatiquement mis à jour dans la vue. Du côté du modèle de la vue, les observables sont créés ou empruntés à un objet du modèle

```
public Observable<…> myLocalObservable = new StringObservable(«…»);
public Observable<…> myLinkedObservable
    = new StringToCharsequenceObservable(model.getXProperty());
```

Ces observables peuvent aussi être de type ArrayList et être utilisé pour une « listView » par exemple ou encore de type booléen pour afficher ou cacher une partie de la vue de façon dynamique.

La liaison modèle de la vue – vue se fait dans les activités grâce à une simple procédure (cf. AppViewActivity) :

```
viewModel = new ModelName(/*paramètres*/);          //on crée le modèle
```

```
setAndBindRootView(R.layout.layoutName, viewModel); //on lie vue et
```

Patron Modèle à Autocontrainte

De plus, la librairie Android Binding permet de vérifier l'intégrité des données. En effet, nous pouvons prévoir un certain contrôle de données en ajoutant une ligne de code au-dessus de l'attribut.

```
@Required(ErrorMessage="You must put the login name!")
public final StringObservable Login;
```

Il existe plusieurs types de contrôle possible. Voici un tableau reprenant ces derniers.

Annotation	Explication
`@Required`	***force l'utilisateur à entrer une donnée***
`@RegexMatch`	***contrôle les données entrées par l'utilisateur à l'aide d'une expression régulière (utile pour les adresses email, les chiffres, etc…)***
`@EqualsTo`	***contrôle les données entrées par l'utilisateur en regardant si la valeur entrée est égale a une autre (utile pour la vérification du mot de passe)***
`@MinLength`	***impose une taille minimum à une donnée entrée par l'utilisateur***
`@MaxLength`	***impose une taille maximum à une donnée entrée par l'utilisateur***

La validation est effectuée en utilisant un "modelvalidator".

```
ModelValidator mv = new ModelValidator(mContext, this, R.string.class);
ValidationResult result = mv.ValidateModel();
```

Selon le résultat de « result », nous pouvons soit effectuer la sauvegarde des données, soit demander à l'utilisateur de vérifier ses données.

Patron Singleton

Nous avons décidé que pour l'histoire «liste de shopping», l'utilisateur ne pourrait avoir qu'une seule liste de shopping sur son téléphone. Nous avons donc utilisé le patron singleton pour réaliser cette liste. Avoir une liste unique utilisant le patron singleton a eu pour avantages:

- Eviter de devoir passer en argument la liste courante à chaque endroit où elle est utilisée.
- Eviter de devoir demander à chaque ajout de produit, dans quelle liste l'utilisateur veut rajouter son produit.
- L'utilisation du patron Singleton à la place d'utiliser une liste statique par exemple, a permis de pouvoir effectuer des tests plus aisément en créant notamment une « fausse » instance (voir shoppingList).

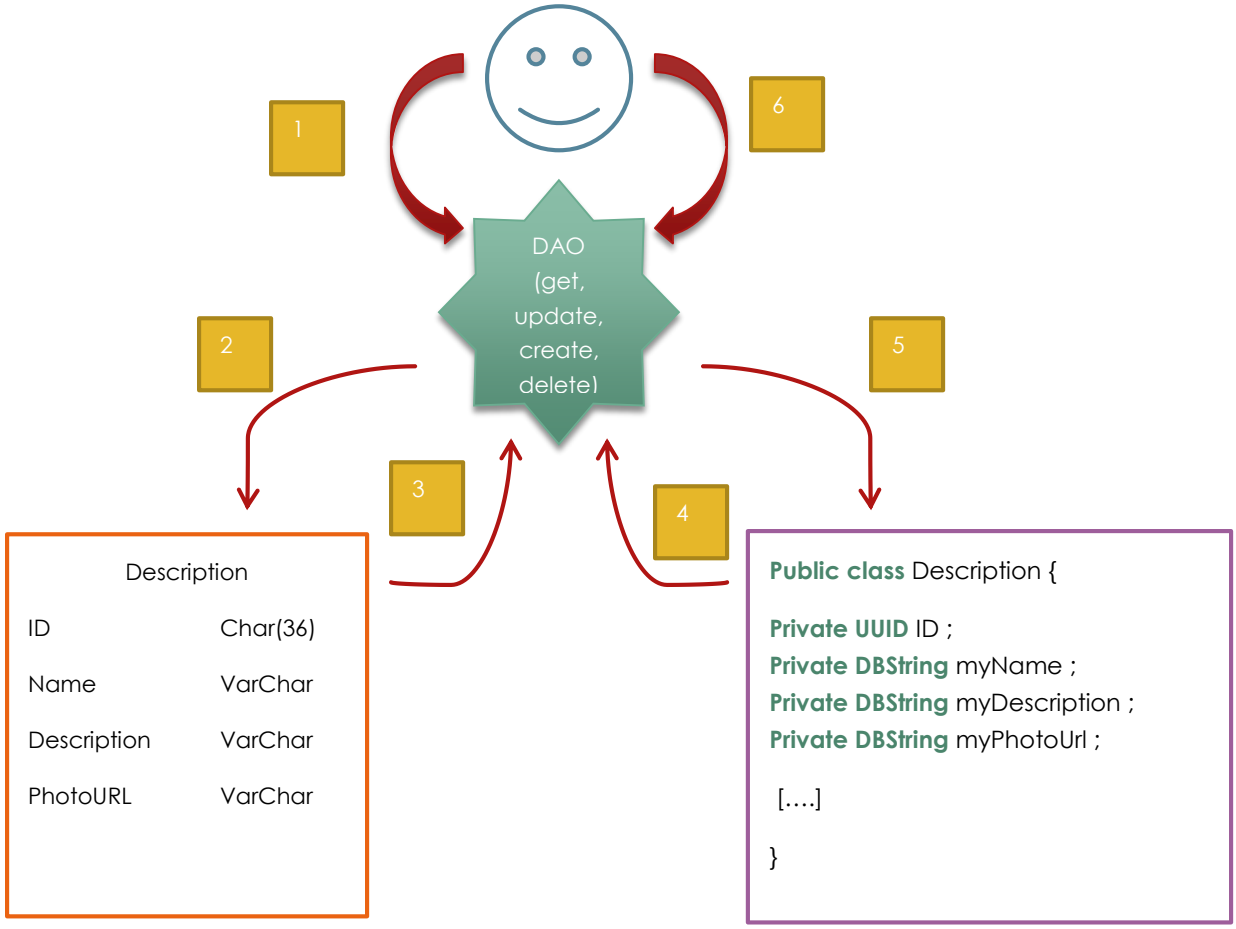
Patron Data Access Object

Ce patron a été utilisé car il permet de faire le lien entre la couche d'accès aux données et la couche métier d'une application.

Il permet de mieux maîtriser les changements susceptibles d'être opérés sur le système de stockage des données, donc, par extension, de préparer une migration d'un système à un autre (BDD vers fichiers XML par exemple).

On a donc une séparation de l'accès aux données et des objets du modèle. En effet, nous allons utiliser un type d'objet pour récupérer les données dans la base de données et utiliser un autre type d'objet pour manipuler ces données.

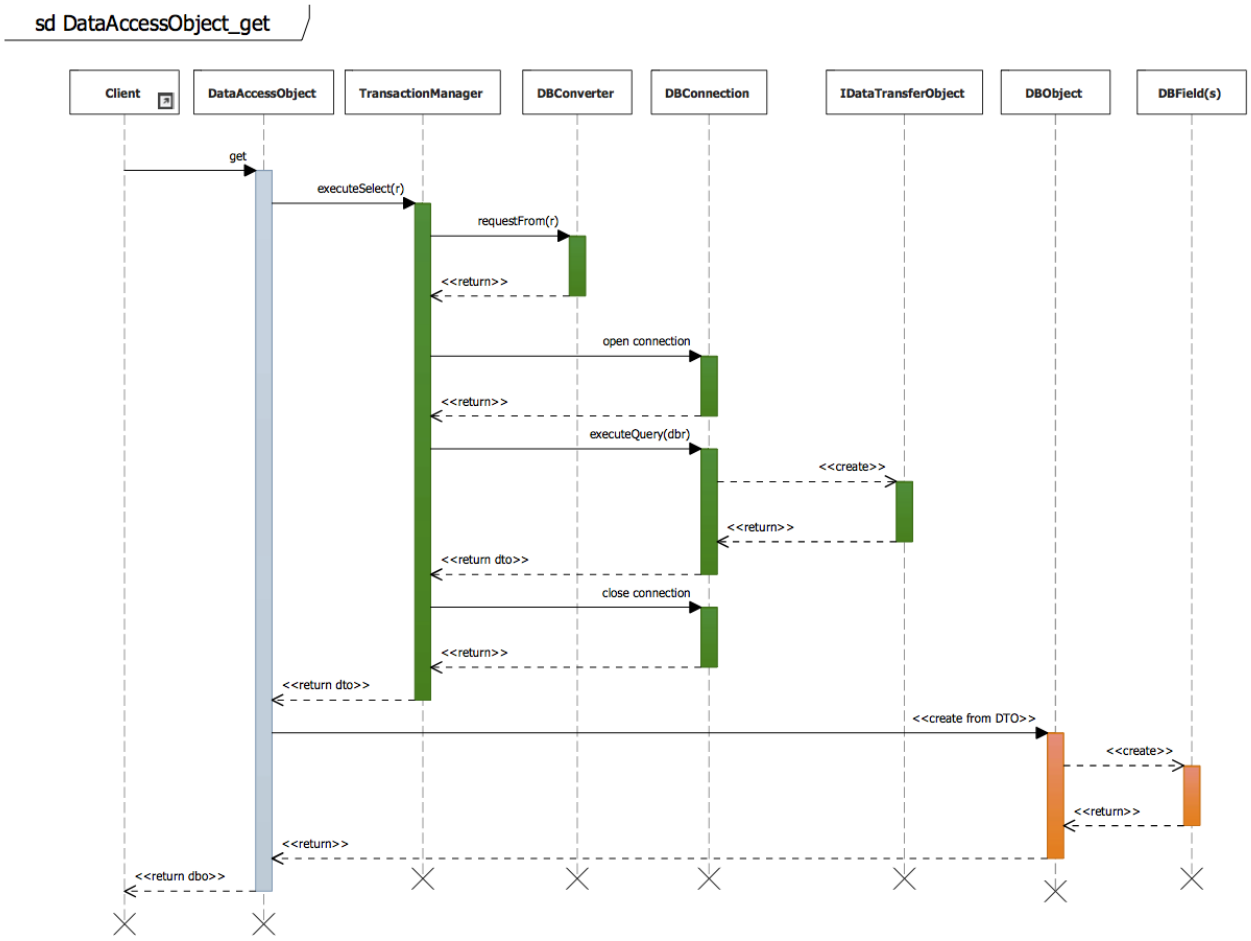
Voici un petit schéma expliquant le fonctionnement (pour les données concernant une description) :



L'application cliente demande un objet (Description) dans notre cas.

1. L'objet **DataAccessObject** récupère cette demande (méthode ***get***) : il s'occupe de créer la requête.
2. Le moteur de base de données (TransactionManager) interprète la requête et retourne le résultat (sous forme d'un DataTransferObject) ; le schéma retourné par le DTO est standardisé par le modèle de données
3. L'objet **DataAccessObject** récupère ces informations.
4. L'objet **DataAccessObject** instancie un objet **Description** à l'aide du DataTransferObject.
5. L'objet **DataAccessObject** récupère cette instance.
6. Enfin, l'objet **DataAccessObject** retourne l'instance de l'objet **Description**.

Voici un diagramme séquentiel pour mieux comprendre le concept :



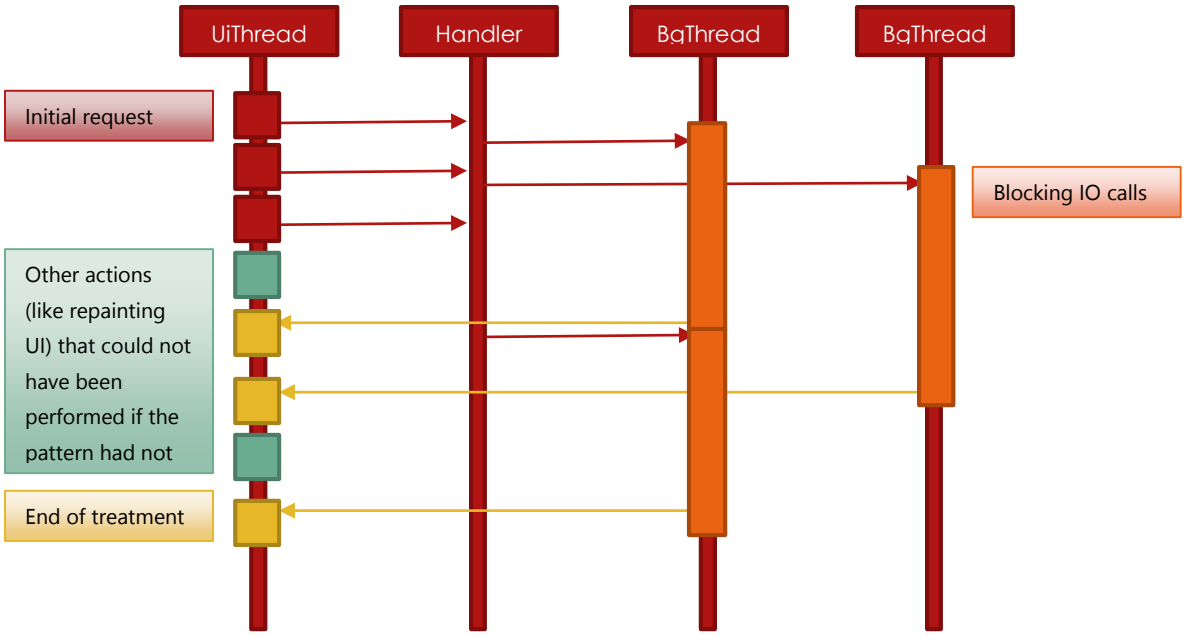
Cependant, cette vision des choses a plusieurs défauts : elle ne marche qu'avec des requêtes exécutées de manière synchrone (appels bloquants) et surtout elle nécessite de télécharger toutes les données avant de retourner un objet.

Dans notre application, nous avons donc apportés quelques modifications à ce pattern. Pour plus d'informations, consultez le document concernant spécifiquement le DAO.

Patron Reactor [Looper/Handler]

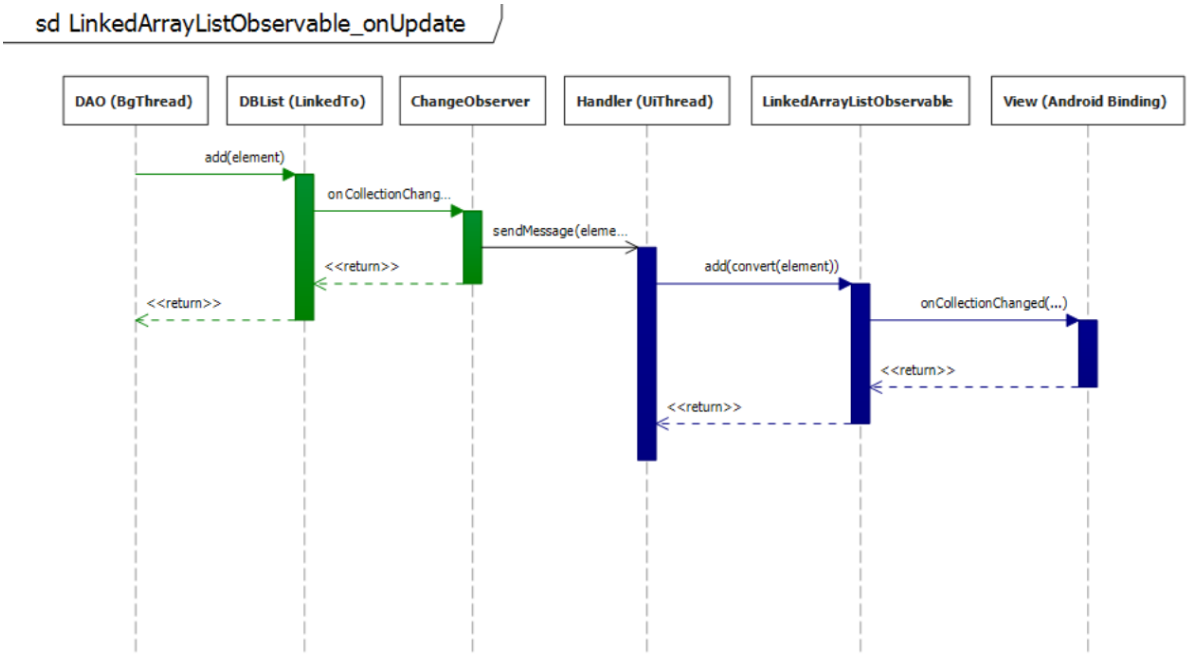
Le patron Reactor est un patron courant qui fournit un système d'entrée-sortie non bloquante. Au lieu d'utiliser un seul thread qui reste bloqué en attendant les opérations d'entrée-sortie se terminent, on assigne un thread ou plusieurs threads additionnels qui sont responsable de gérer les entrées-sorties (thread pool).

Lorsque toutes les opérations d'entrée-sorties sont finies, le thread génère un évènement pour qu'un autre thread commence à gérer les données reçues. Ceci est utile lorsque nous avons plusieurs connexions à gérer. On n'est pas forcé de dédier un thread pour chaque connexion, ce qui pourrait consommer beaucoup de ressources si le nombre de connexion est conséquent, c'est pour cela qu'on utilise un thread pool et le patron Reactor.



Le View Handler est un patron de vue qui aide à gérer les vues de notre model. Il permet au client d'ouvrir, manipuler et fermer les vues. Il coordonne aussi les dépendances entre les vues et organise leur mise à jour.

Dans notre cas, lorsque les changements sont fait dans le thread gérant la base de données (BgThread), nous avons un autre thread qui met à jour la vue du model (UiThread) Voici un diagramme de séquence expliquant son fonctionnement dans notre application :



Les deux threads utilisent le patron Reactor pour communiquer. Dans un sens, le modèle signale au BGThread les changements, et le BgThread s'occupe d'appliquer ces changements à la base de données. Dans l'autre sens, quand il y'a des changements sur le BgThread, on a le pattron Handler qui s'occupe de mettre à jour la vue via le UiThreads.

Patron d'initialisation tardive

Dans le DataAccessObject, plusieurs listes synchronisées ne sont créées que lorsqu'elles sont réellement utilisées. Pour ce faire, on se base sur le système interne de la machine virtuelle Java pour l'initialisation tardive des champs statiques des classes.

Ce design pattern est important pour rendre le chargement initial de l'application plus rapide en évitant de faire des calculs inutiles : en effet, il est probable que certaines données ne seront jamais utilisées par la suite par l'utilisateur, il est donc totalement absurde de faire ces calculs préemptivement comme le feraient beaucoup de développeurs peu attentifs.

Patron Multiton

Ce design pattern est utilisé au niveau de la classe DBConverter : en effet, une liste de convertisseurs est proposée à l'utilisation ce qui permet de les réutiliser à plusieurs endroits. Ce design pattern ressemble finalement assez fort au pattern singleton (qu'il a d'ailleurs remplacé dans notre application dans ce cas particulier).

L'avantage de ce pattern est qu'il conserve la simplicité et l'unicité du pattern singleton, mais permet malgré tout plusieurs variations connues mais limitées (comme par exemple plusieurs formats de fichiers, ou différentes versions d'un standard).

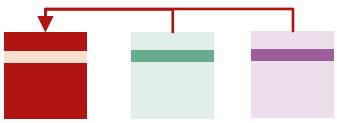
Dans notre cas, il permet de représenter un ensemble de standards de DML (Data Manipulation Language) qui sont supportés par notre application (MySQL et SQLite) mais qu'on pourrait étendre davantage (par exemple en ajoutant le support de XPath).

Patron des commandes

Le design pattern de commande a pour objet de remplacer certaines fonctions par des objets représentant l'action en question. Cela permet de passer une commande en argument dans le constructeur d'un objet (comme dans le cas d'un ItemList_VModel). Cela permet aussi beaucoup de souplesse dans la représentation des commandes (comme la composition, la mise en série, …).

Le désavantage du patron de commande est évidemment une consommation mémoire légèrement accrue, et une petite perte de performance à l'exécution. Comme notre application est extrêmement fluide et ce même dans l'émulateur, ce n'est pas un souci pour nous.

Patron des méthodes exemplatives



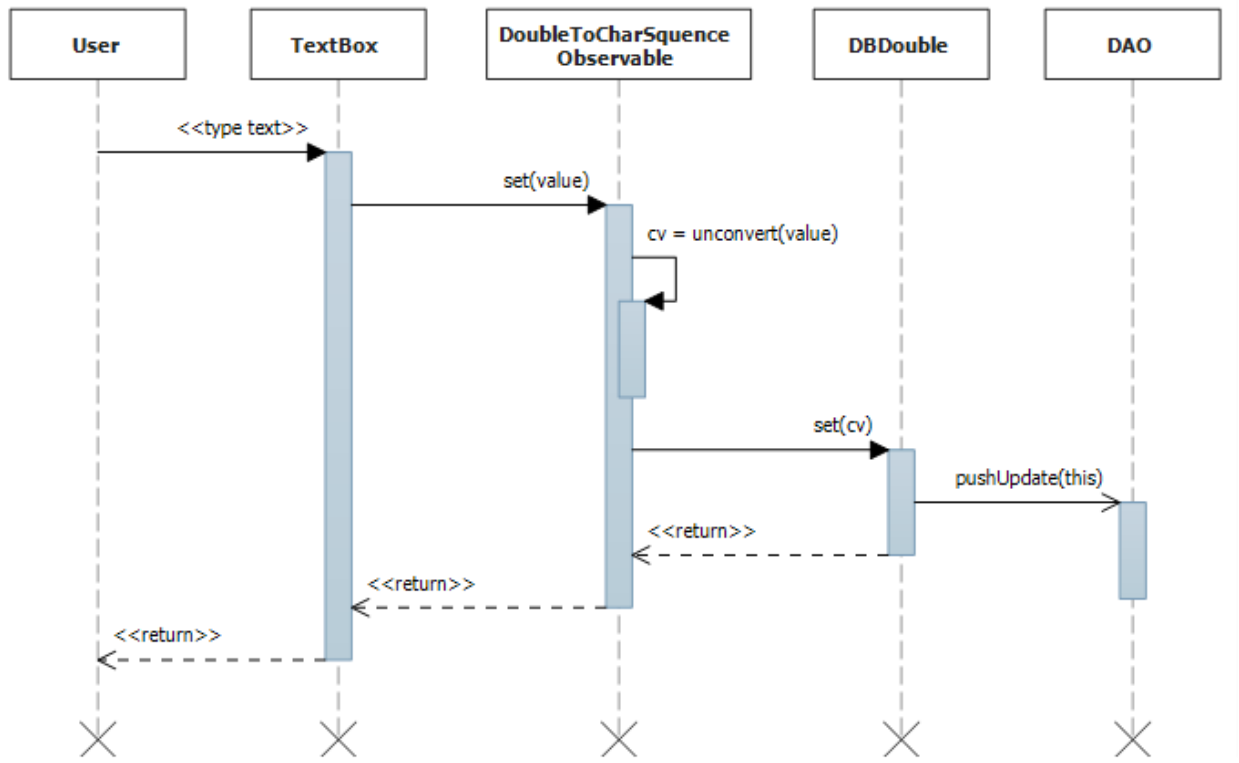
Ce design pattern permet de déplacer partiellement un algorithme vers une classe parent tout en laissant des points de variations aux classes filles. C'est tout particulièrement utile dans la classe AppViewActivity, qui possède deux niveaux de variabilité qui peuvent demander certains calculs : nous avons ainsi définis des méthodes abstraites pour laisser aux classes filles d'agir sur le constructeur avec des informations qui ne sont disponibles qu'après le début de celui-ci exécuté. Cela permet d'éviter de la duplication de code.

Patron du verrou bloquant lecture-écriture

Dans notre application, nous faisons régulièrement appel à des verrous bloquant lecture-écriture. C'est notamment le cas des endroits où certains objets peuvent être lus par plusieurs threads en même temps mais ne peuvent être modifiés par deux sources simultanées. Dans ce cas, plutôt que d'utiliser une synchronisation de la classe entière, on n'acquiert de lock que pour lorsque cela est nécessaire et par ressource au lieu de par ensemble de ressources. Cela complique un rien le code mais cela évite des blocages entre threads superflus.

Patron des propriétés liées

Dans notre application, nous avons des « Observables Liés » qui convertissent un type de donnée en un autre dans les deux directions, pour s'assurer que l'affichage (ex : une boite de texte) et le modèle reste cohérents entre eux. Exemple d'intérêt : « DoubleToCharSequenceObservable » et « StringToDrawableObservable ».



Patron d'adaptation

Le patron « Adapter » permet de transformer un objet extérieur à notre projet et obéissant à une certaine interface de s'intégrer dans notre projet en « gommant » les différences par rapport à l'interface choisie dans le cadre de notre projet.

Dans notre cas, le patron d'adaptation est utilisé pour les DTO (ex : CursorToDTO) car la base de données SQL et la base de donnée SQLite ont des interfaces différentes et on ne veut pas se soucier de cela dans la couche DAO.

D'une activité à l'autre

Introduction

Lorsque l'on conçoit une application Android, on se retrouve rapidement confronté au problème de la communication de données entre activités. Par défaut, Android ne propose pas de modèle d'échange de donné qui soit vérifiable à la compilation : c'est à l'activité parente et à l'activité enfante de s'échanger les bonnes données sur base de la documentation générée.

Communication par contrats

Cette méthode de communication n'est pas du tout une bonne solution, en raison du « potentiel de problème » qu'elle représente. Afin de rendre les choses meilleures, il a été décidé de faire communiquer les activités entre elles via un système de « contrat » typé.

Chaque activité capable de recevoir des arguments en entrée et/ou d'en retourner en sortie doit hériter de la classe générique InputOutputActivity qui précise quel type est attendu en entrée et quel type est renvoyé en retour.

Ensuite, pour démarrer une activité, nous avons surchargé la méthode startActivity pour ajouter des fonctions prenant un type générique d'activité d'entrée-sortie, un objet de l'argument d'entrée attendu par l'activité et d'un callback gérant le type de sortie prévu de l'activité.

Cela assure qu'il ne peut pas se produire d'erreur d'argument mismatch entre un appel d'activité et sa création.

Détails de l'implémentation

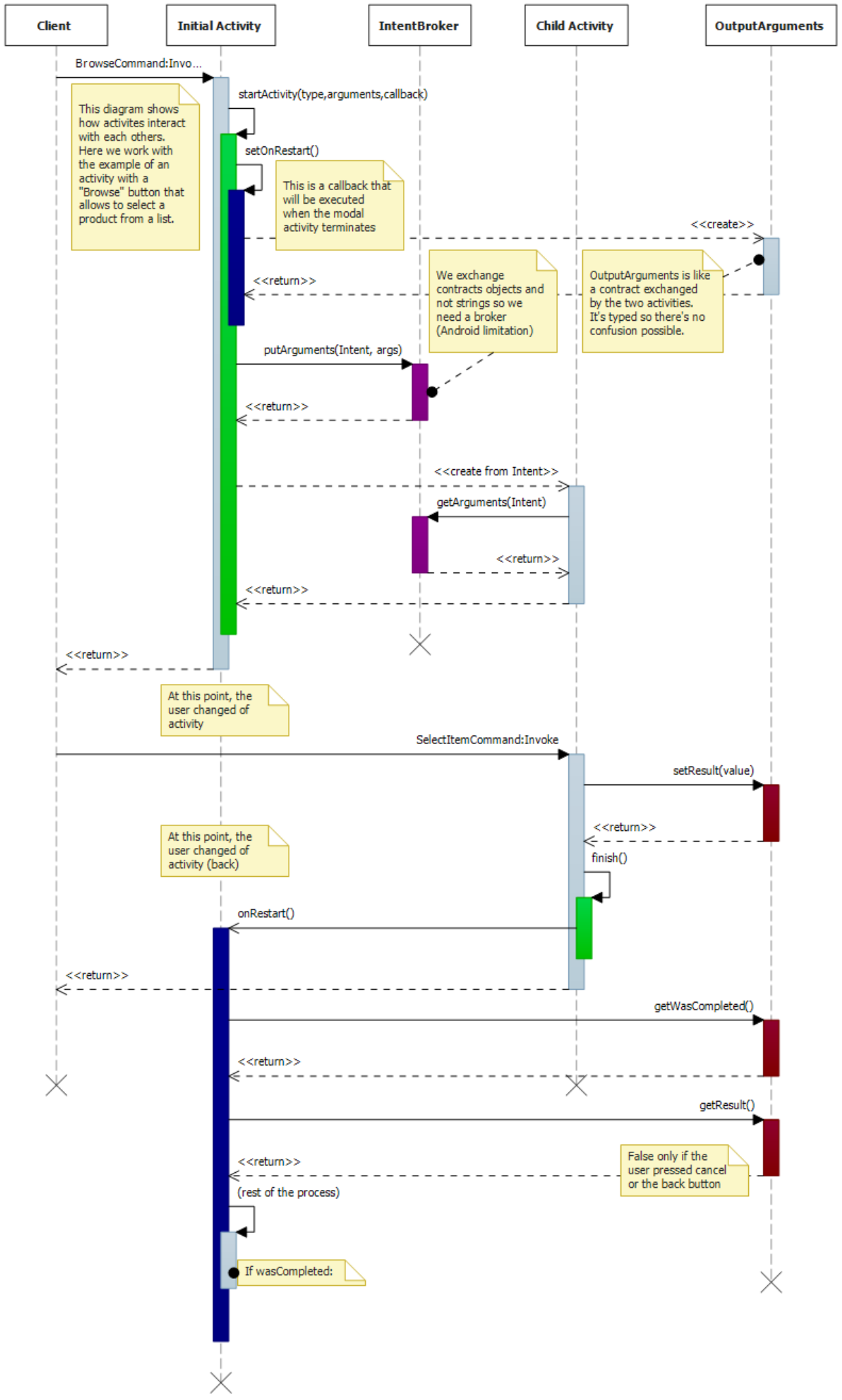
Par défaut, il n'est pas possible de passer des objets à une autre activité, ce qui rend le système de contrat difficile à implémenter. Pour s'assurer de son fonctionnement, nous avons donc développé un Helper (IntentManager) qui s'occupe de « passer les arguments » d'une activité à l'autre.

L'activité parente obtient un « identificateur » temporaire pour les objets qu'elle souhaite passer en retour (InputType et OutputArgument<OutputType>) et cet objet est stocké dans une table globale le temps qu'il soit demandé par l'activité suivante à partir de l'ID qu'elle aura reçu via son Intent.

Il est aussi intéressant de noter que si l'utilisateur appuie sur Back ou sur un bouton Cancel, la valeur de retour est bien entendu nulle, mais un champ particulier (*wasCompleted*) indiquera que l'action de l'activité précédente n'a pas été menée à son terme.

Contrats types génériques.

Dans l'application, il y a deux contrats types : le Chooser (qui prend une liste de IStoreObject en argument et retourne un élément de la liste tel que sélectionné par l'utilisateur) et le Viewer/Editor, qui prend un élément et l'affiche à l'écran (avec parfois possibilité de modification).


Client
Initial Activity
IntentBroker
Child Activity
OutputArguments
BrowseCommand:Invo...
This diagram shows how activites interact with each others. Here we work with the example of an activity with a "Browse" button that allows to select a product from a list.
startActivity(type,arguments,callback)
setOnRestart()
This is a callback that will be executed when the modal activity terminates
<<create>>
<<return>>
We exchange contracts objects and not strings so we need a broker (Android limitation)
OutputArguments is like a contract exchanged by the two activities. It's typed so there's no confusion possible.
putArguments(Intent, args)
<<return>>
<<create from Intent>>
getArguments(Intent)
<<return>>
<<return>>
<<return>>
At this point, the user changed of activity
SelectItemCommand:Invoke
setResult(value)
<<return>>
At this point, the user changed of activity (back)
finish()
onRestart()
<<return>>
getWasCompleted()
<<return>>
getResult()
<<return>>
False only if the user pressed cancel or the back button
(rest of the process)
If wasCompleted:

Android Binding

Introduction

Après une recherche sur les possibilités de binding natives d'Android, nous avons décidé d'utiliser une librairie open source supplémentaire permettant de gérer la liaison de données entre la vue et le modèle nommée Android Binding. Celle-ci est basée sur le concept d'observable et permet de pouvoir découpler les activités des vues. Elle favorise l'utilisation du patron MVVM, idéal pour une application Android devant principalement afficher des données.

Avantages

- Cela permet aux développeurs de gagner du temps lors de l'écriture des vues et des interactions avec l'utilisateur qu'elles peuvent posséder. En effet, tout ce qui est visible à l'écran se trouve dans le code XML de la vue et tout ce qui est procédural ce trouve dans le code Java du modèle.

- Cela permet d'améliorer la fiabilité du code. En effet, après qu'un utilisateur ait rajouté/supprimé un élément à une liste ou ait modifié un de ceux-ci, la mise à jour de la vue n'est pas automatique mais demande un appel de fonction spécifique. Cela signifie que l'oubli de quelques lignes de code peut vite poser de subtils problèmes de désynchronisation qui ne sont pas toujours évidents à résoudre. Cela suppose aussi un couplage fort entre le modèle et la vue, ou l'implémentation systématique d'un patron d'observables. Android Binding permet de mettre en œuvre cette dernière solution simplement.

- Cela permet également de faciliter la maintenance et le développement de l'application. En effet, le binding permet d'éviter l'écriture de beaucoup de code redondant et peu intéressant, pour lequel on finit vite par perdre toute attention.

Désavantages

- L'ajout d'une librairie de binding peut ralentir légèrement l'exécution du programme et augmenter sa consommation mémoire (chaque observable ajoute de l'overhead) mais c'est totalement négligeable dans le cas de notre application car elle ne risque pas de pousser dans ses limites les ressources d'un smartphone actuel (raisons : notre interface graphique très simple, gros du travail effectué côté serveur, …).

On fait donc un compromis entre « lignes de code à la compilation » et « mémoire à l'exécution».

Conclusion

Nous en concluons donc que, dans le cadre de ce projet, l'utilisation de cette librairie pourra nous permettre d'aller plus vite dans nos développements et de commettre moins d'erreurs.

Pour plus d'information sur cette technologie :

http://www.codeproject.com/Articles/145203/Android-Binding-Introduction

http://code.google.com/p/android-binding/

Base de données

Choix d'une nouvelle base de données

Nous avons commencé l'implémentation d'une nouvelle base de données pour cette itération afin de mieux correspondre à notre implémentation orientée objet. Cela nous permet d'avoir un schéma identique entre notre modèle et notre base de données, et permet donc une automatisation de la mise à jour du modèle depuis la base de données en respectant l'héritage des classes.

En outre, cela nous permet de ne pas dépendre des choix des autres groupes pour implémenter une nouvelle fonctionnalité.

Le but de cette base de données «orientée objet» est de simplifier l'utilisation des concepts d'orienté objet dans le domaine relationnel.

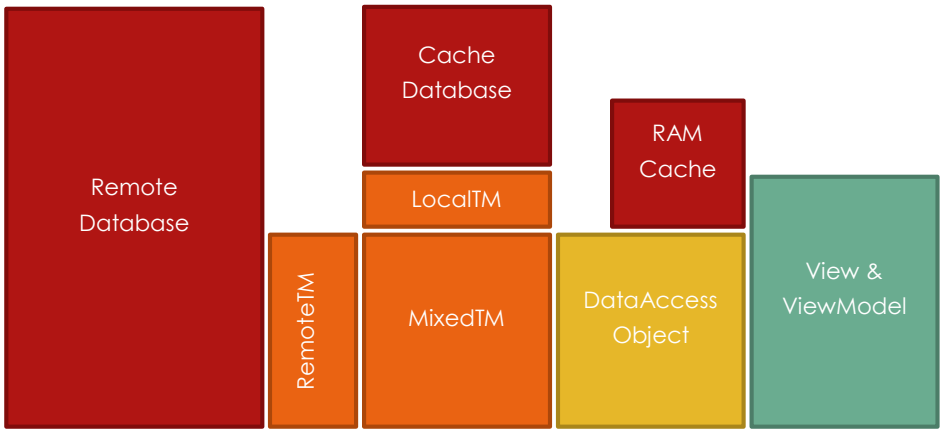
Notre analyse nous a montré que les éléments de la base de données seraient utilisés essentiellement de deux façons différentes : soit sous forme de liste, auquel cas on n'utilise que certaines des propriétés, communes à tous les objets ; soit sous forme détaillée pour un objet unique, où l'on regarde toutes les informations sur cet objet. On peut rendre les objets de la base de donnée "listables" en les faisant hériter d'une superclasse commune, appelée Description, qui peut être employée sans devoir se préoccuper des propriétés spécifiques de chaque objet réel.

Pour permettre l'éventuelle implémentation d'un cache local en plus de la base de données, on a également implémenté différentes interfaces Java, permettant d'accéder à une ligne de la base de données en lecture ou en lecture/écriture. Ces interfaces seront ensuite implémentées par des classes permettant cette implémentation de plusieurs bases de données.

Lorsqu'un objet Description voit une de ses propriétés observables modifiées, il notifie automatiquement la base de données du changement. De même, lorsqu'un tel objet vient d'être créé par l'application, il enverra automatiquement une requête à la base de données pour y être ajouté.

Gestion du cache

Notre base de données fonctionne selon un modèle de cache ; nous discutons plus en détails dans la partie sur le DataAccessObject sur la manière dont le cache est mis à jour, mais voici comment on peut visualiser les interactions entre les différents niveaux de cache et l'application (les objets contigus interagissent) :



Data Access Objects

Introduction

Pour les besoins de notre projet, nous avons développé une interface avec la base de données. Pour les besoins du projet (connexion lente à la base de données et faible débit dans les communications) et de l'utilisation mobile (connexion Internet intermittente ou inactive), notre application est prévue pour fonctionner en mode « hors ligne ».

Fonctionnement général

Le modèle fonctionne selon un système proche du DAO classique. Un DataAccessObject s'occupe d'envoyer des requêtes de recherche, de création et de modification à la base de données via son TransactionManager. Le gestionnaire de transaction convertit ensuite les requêtes en SQL. Le TransactionManager exécute les requêtes de manière asynchrone sur un thread secondaire pour ne pas bloquer le thread principal. Cela permet d'être très réactif et de fonctionner sur Android 4 en plus d'Android 2.

Mode hors ligne

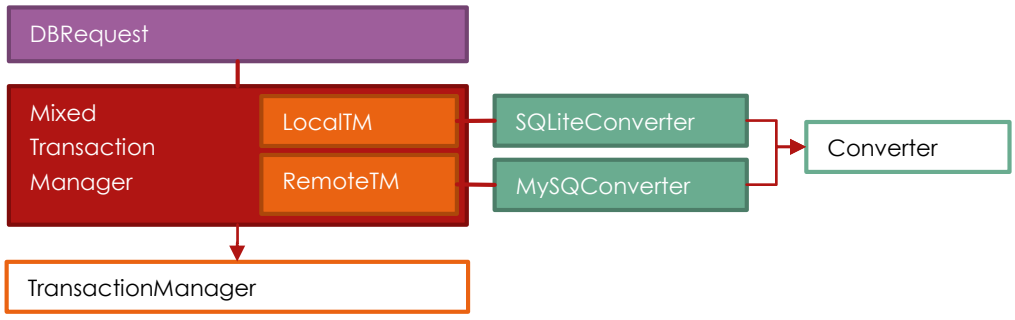
Pour faire fonctionner l'application hors ligne et accélérer les chargements, une base de données locale SQLite est utilisée comme cache. Cette base de données est initialement vide lorsque l'on installe l'application mais utilise toutes les données téléchargées depuis la base de données principales pour se remplir.

Cela signifie qu'il est possible par après de revenir sur tous les écrans par lesquels on est déjà passé (vu que les données auront été téléchargées en local). Par ailleurs, quand la connexion Internet n'est pas accessible, les requêtes d'éditions et d'insertions sont stockées dans un backlog pour une exécution délayée.

Language Independent Queries

La manière dont cela est gérée est la suivante: les requêtes sont représentées par des objets similaires aux Monads de LINQ (*Language Independant Query Language*) de Microsoft. Comme dans le cas de LINQ, il existe des convertisseurs capables de convertir les requêtes indépendantes en requêtes spécifiques. Dans le cadre de notre projet, deux langages sont supportés : MySQL et SQLite.

Le gestionnaire de requête (Data Access Object) est lié à un gestionnaire de transaction (TransactionManager). Dans le cas où l'on travailler sur deux bases de données, il s'agit d'un MixedTransactionManager, c'est-à-dire d'un gestionnaire de transactions qui prend une base de données maitresse et une base de données cache. La base de données maitresse est un gestionnaire de base de données distante (MySQL) tandis que la base de donnée cache est une DB locale. On pourrait bien entendu faire l'inverse mais ça n'aurait aucun sens.



Base de données orientée objet

Pour coller au plus proche de ce qu'on utilise au niveau de Java, on utilise une base de données orientée objet. Chaque objet possède une « adresse mémoire » (identificateur) unique appelé Globally Unique Identifier (GUID) qui l'identifie de manière unique à travers toutes ses interfaces (c'est à dire toutes les tables dans lesquelles il se retrouve). Cet identificateur permet de créer des objets de manière distribuée (mode hors ligne) sans se soucier d'éventuels conflits.

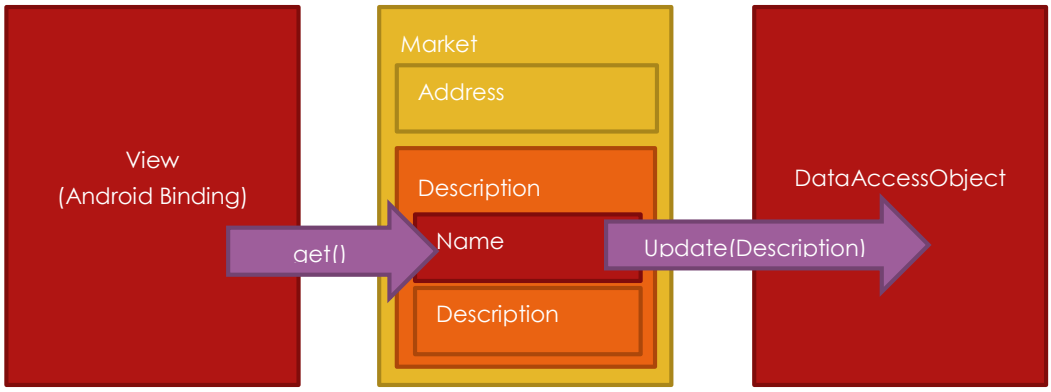
Pour revenir au schéma de la base de données, pour prendre un exemple concret, un supermarché est à la fois une description (Nom, Description, PhotoURL) et un supermarché (Adresse, Longitude, Latitude…).

Notre application ne télécharge les données que lorsqu'elles sont réellement utilisées. L'avantage de la séparation en interfaces, c'est que l'on peut réutiliser le code (gestion des listes…) et que lorsqu'on n'a besoin que des données d'un type (comme Description) on ne doit pas télécharger les données propre à un supermarché.

Chargement sur demande

L'application est prévue pour que lorsqu'une donnée est demande à un observable des informations qui ne sont pas encore chargée, leur chargement est mis en liste d'attente et planifiée pour exécution dans le thread secondaire. C'est ce qu'on appelle chargement sur demande.

Voici un exemple de ce qui se passe quand on affiche le nom d'un marché dans une liste mais que la donnée n'est pas encore disponible :



Chargement asynchrone et des listes

Il est intéressant de noter que les listes disposent à elles-seules de pas mal d'observables permettant de savoir si elle est vide ou pas, si elle a déjà eu le temps de charger les données, si jamais il reste des données à charger (*en effet, pour accélérer les choses, la plupart des listes ne sont chargées que par blocs de quelques éléments, et un bouton permet de demander le téléchargement de la suite*).

Toutes ses fonctionnalités asynchrones ne fonctionnent que parce que toute l'application est basé sur des observables liées à la vue, ce qui nous rends très content du choix d'Android Binding.

Mise à jour automatique des listes

Quand un élément est ajouté ou retiré de la DB, il est automatiquement retiré des listes toujours en vies sur le téléphone (en effet, les requêtes supportent aussi un mode d'évaluation « local » prévue à cet effet).

Diagrammes

Diagramme des classes

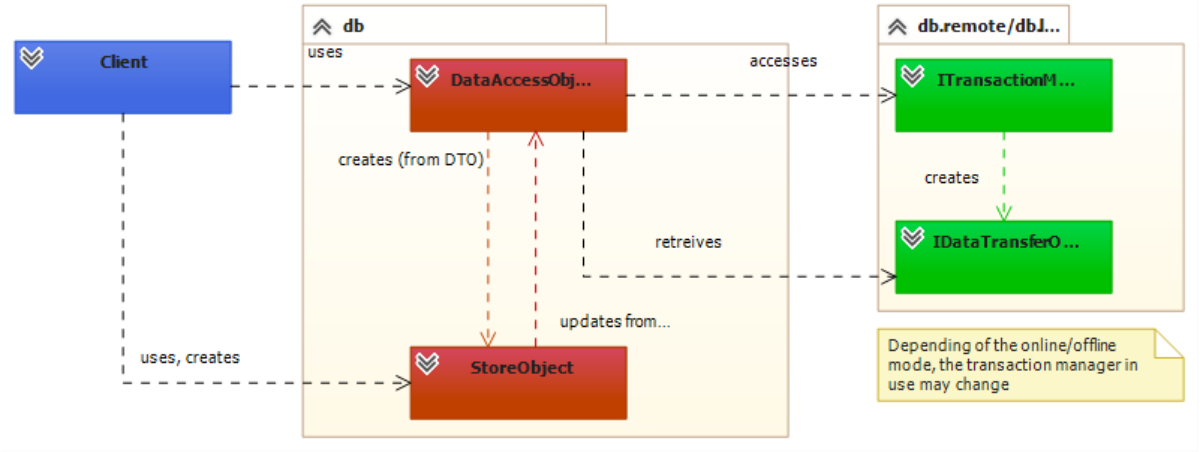
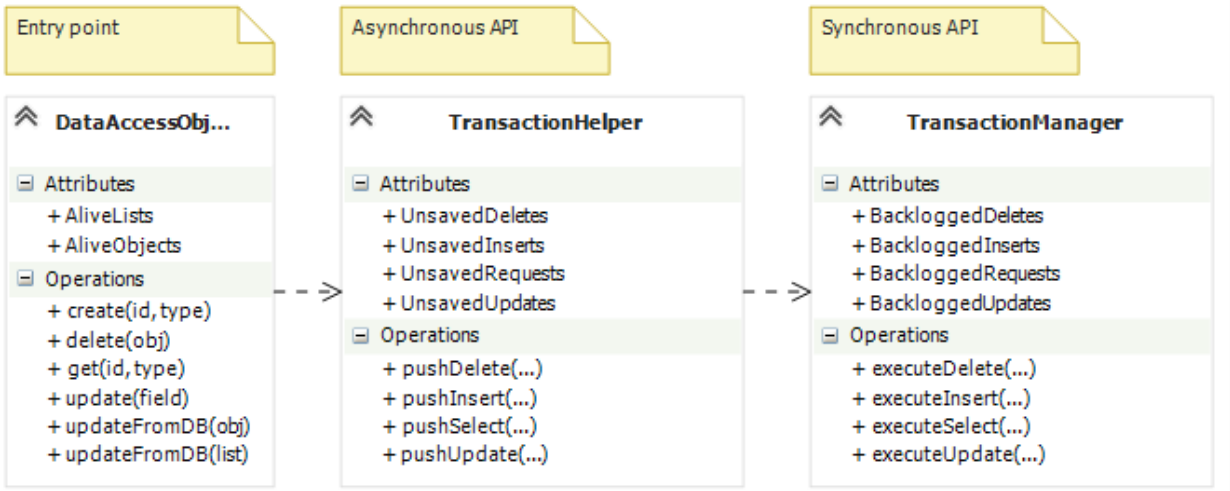
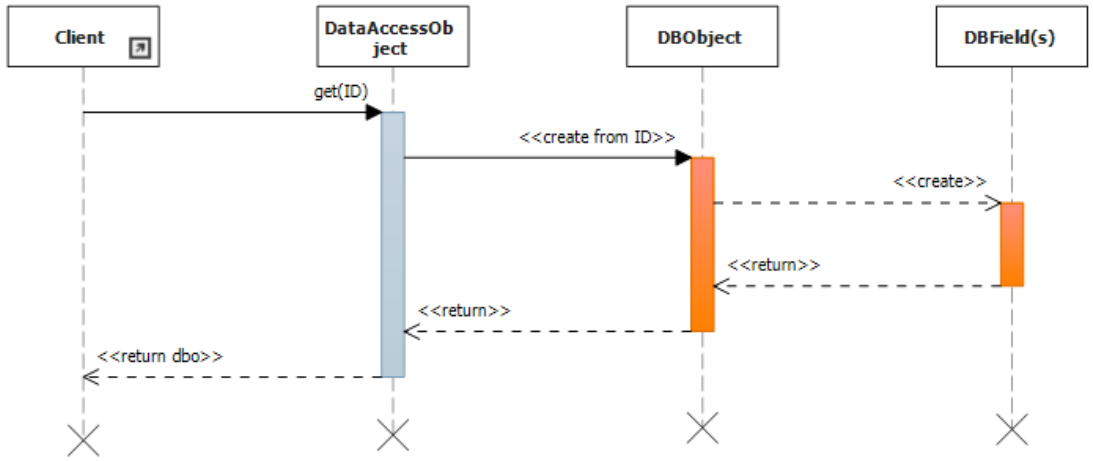


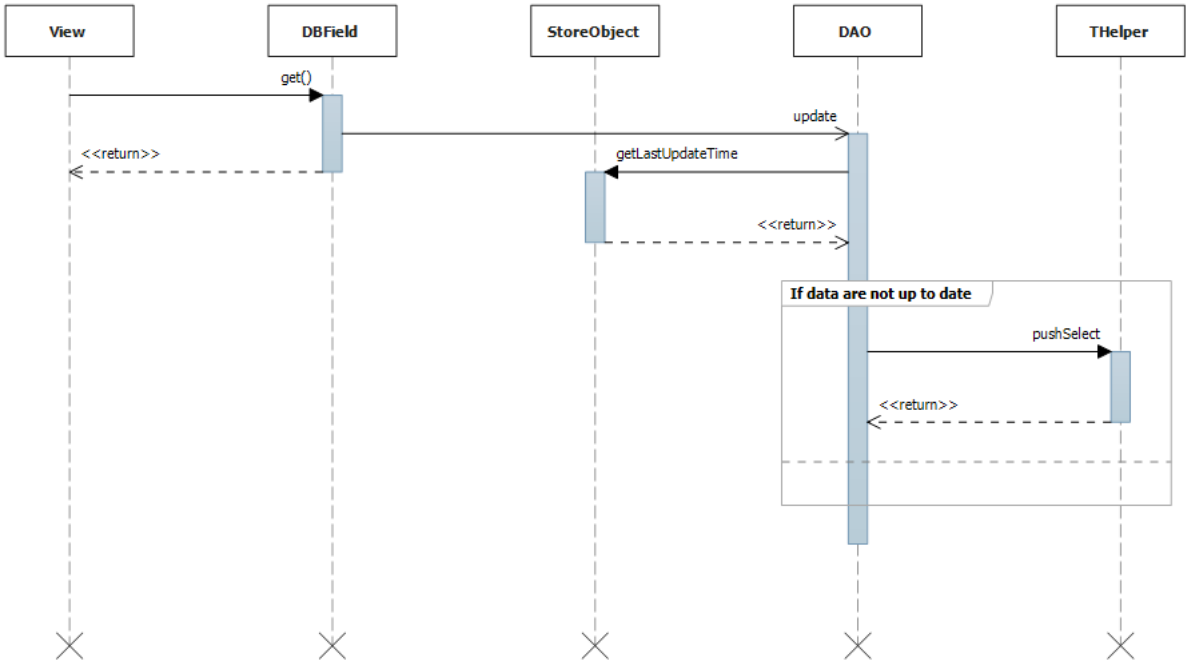
Diagramme de classe du DAO générique



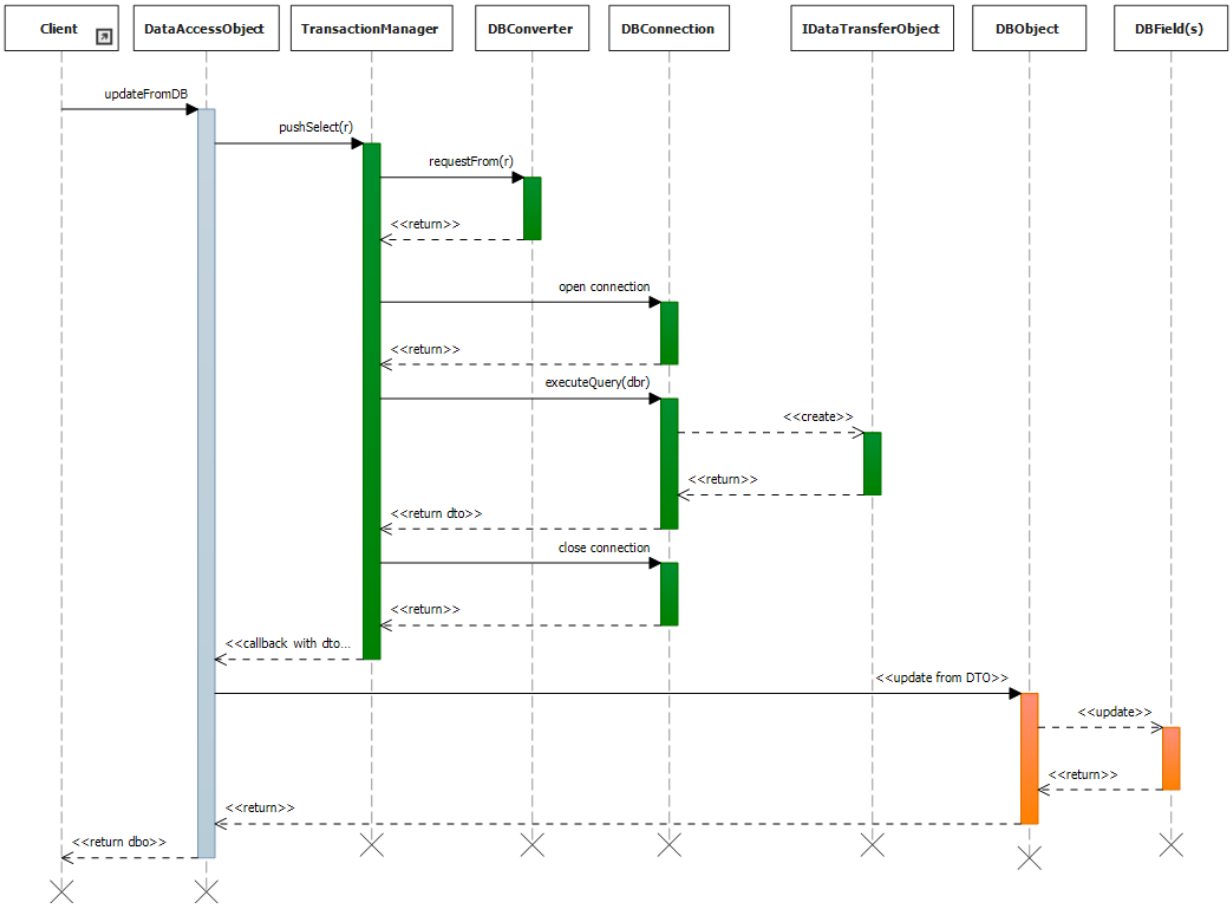
Séquence « get by ID »



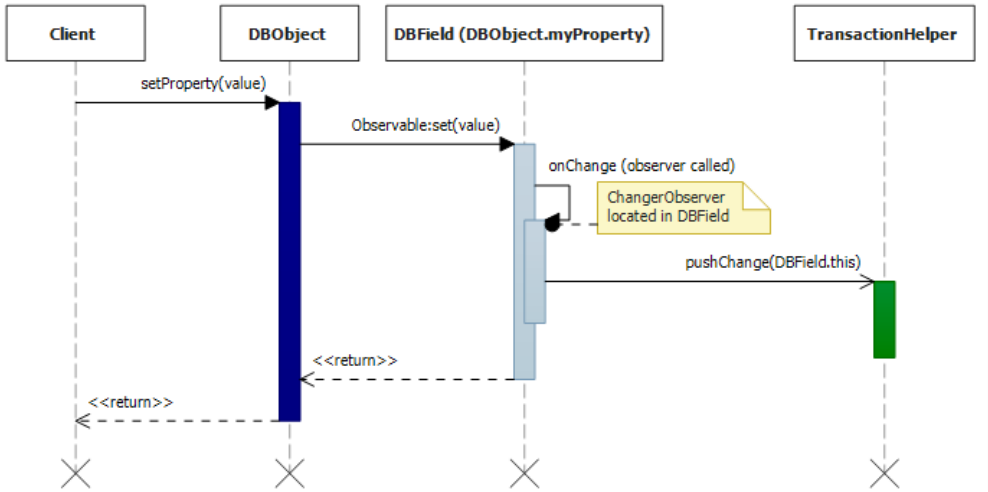
Séquence « download-on-demand »



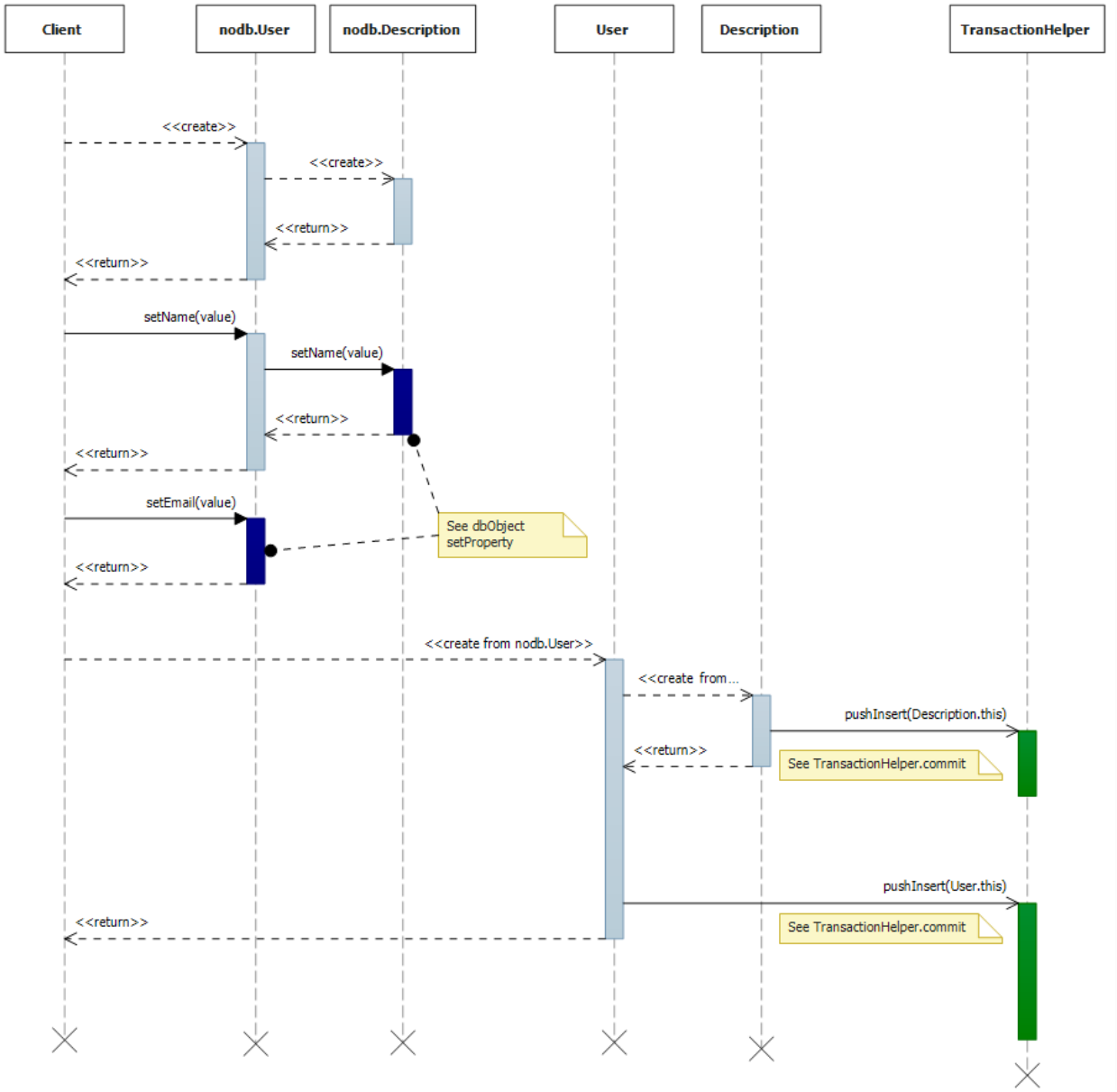
Séquence « update from DB »



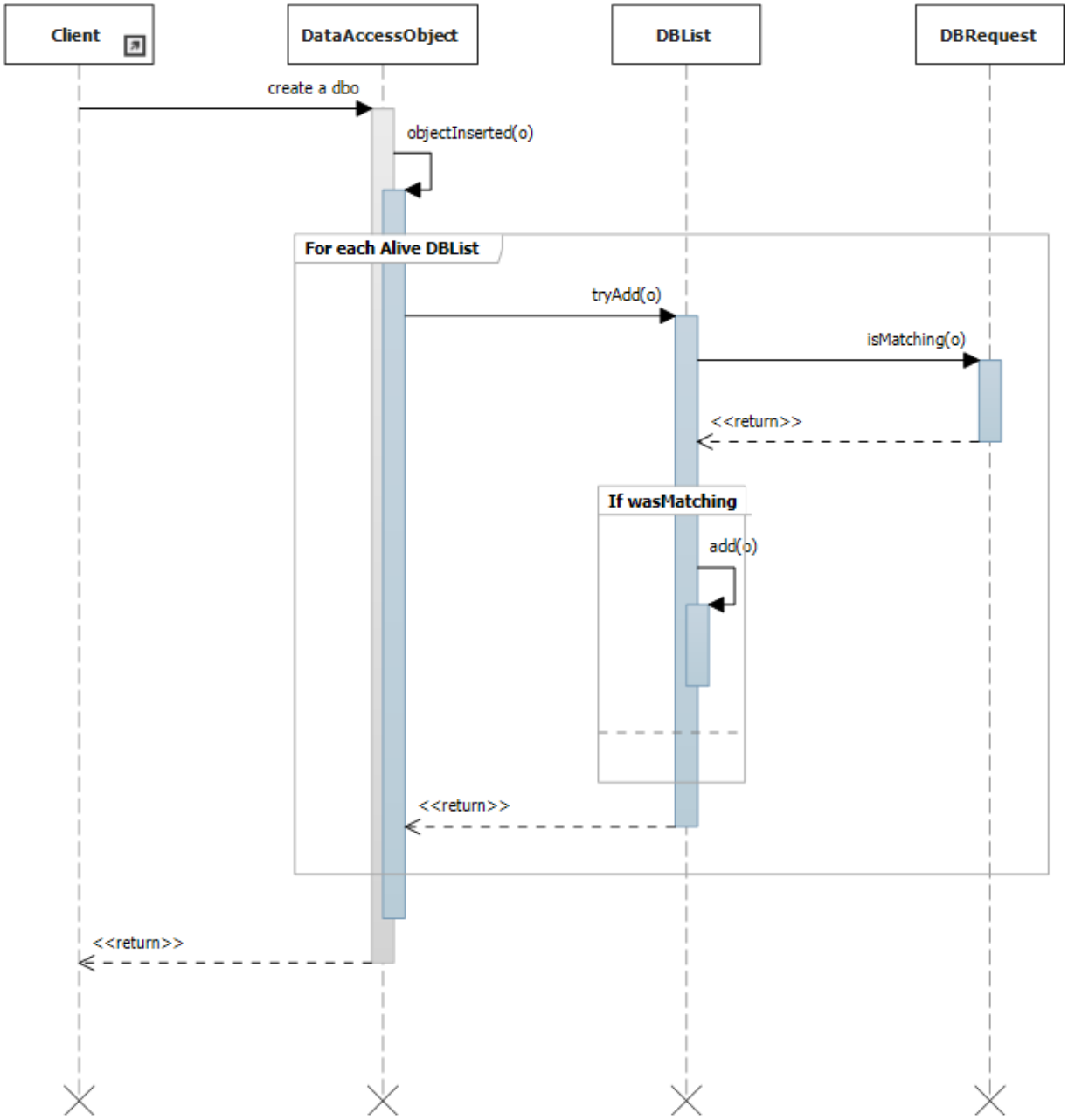
Séquence « update to DB »



Séquence « insert into DB »



Séquence « list update on insert »



Séquence « remove from DB »

Semblable à certains diagrammes précédents: on envoie une requête de suppression à la base de donnée et on supprime l'élément de toutes les listes en mémoire qui le contiendraient encore.

Code auto-généré

Introduction

Dans le cadre de notre projet, nous auto-générons tout le code qu'il est possible d'auto-générer à partir du schéma de la base de donnée :

- CREATE TABLE (SQL Database Schema)
- DAO-enabled model (app.models.*)
- DAO-disabled model (app.models.nodb.*)
- Common read-only and read-write interfaces (app.models.interfaces.*)

Nous avons écrits le générateur de code nous-même (*Razor/ASP.NET MVC 4*), et le format d'entrée du générateur a été choisi pour répondre au mieux à nos besoins (simples et efficace) est le suivant (*extrait*):

```
Description ( Name as String, Description as String, PhotoUrl as String)
Product extends Description ( PricePerUnit as Double )
Recipe extends Description ( PricePerPerson as Double, MinPerson as Double, Instructions as String )

RecipeProduct ( Recipe as Recipe, Product as Product, Quantity as Double)

User extends Description ( Email as String, Password as String )
UserComment extends Description ( User as User, Target as Description, Rating as Double )
```

Avantages

- On évite les bêtes fautes en écrivant le code une seule fois, dans des canevas.
- On évite les « exceptions », on pense générique dès le premier instant
- On gagne du temps quand il faut changer le modèle par après (nouvelles fonctionnalités)

Désavantages

- On perd du temps quand il faut des fonctionnalités particulières
- On ne peut pas toucher aux fichiers auto-générés
- Parfois difficile à comprendre (haut-niveau d'abstraction)

Conclusion

Au final, ce système nous a fait gagner beaucoup de temps car il nous a permis d'éviter de devoir maintenir similaires manuellement les interfaces et les instances des objets du modèles, tout en permettant d'effectuer des corrections à de nombreux endroits du projet depuis un seul point.

Le code du générateur n'est pas inclus dans le projet mais peut sans difficulté être demandé par mail si jamais il vous est nécessaire de regarder comment il fonctionne.

Security Report

Weakness

Inline database connection

The application has several issues: firstly, the database is used directly on the smartphone which mean that the admin password to the database is exposed in the source code of the application and is transmitted over the network during the connection phase.

Public access to database

Users can perform DOS attacks on the database by requesting many data or performing many connections. There's no proxy handing such attacks. Similarly, there's no way to "keep data" out of control of a malicious attacker or someone that want to fork our service (and therefore copy all data it already contains).

Possible Workarounds

Using a web service to perform load balancing and protecting data through authentication and user right management. But this was not planned by the project supervisors, so this is an accepted vulnerability.

Storage of passwords

Initially, all passwords stored in the database in clear type. And since the interaction with the database occurs through unprotected channel, there is a risk of their use by an attacker. So, following safety options are existed:

1. **Using the tool built-in database**. In MySQL (if installed on Unix) there exists a function "encrypt('str')" that uses Unix algorithm of encrypting. Also exist such functionality as encode (str, pass) etc. But that the application should be independent of the type of database or of the platform in which it's installed. So, this way is not for us.

2. **Using Java functionality for security**. There are many ways here. From one line using, for example, android.util.Base64 up to possibility write AES or DES or even RSA. But security is not the main goal of our project. And the main principle of a system of data protection - the costs of the system of protection should not exceed the value of the protected information. So, this way is not for us too.

3. **Using hash-function**. There is a standard library in java: java.security.MessageDigest (Uses a one-way hash function to turn an arbitrary number of bytes into a fixed-length byte sequence. More information can be found here http://developer.android.com/reference/java/security/MessageDigest.html) that can use MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512. So, there we chose it.

Password hashing performs a single static function. Since it is one-way function, when you sign compares the hash of the password and stored in the database. Here there are protected passwords, as they appear in the database:

ID	Email	Password
c5efccc9-eab6-4024-836e-ebdfa5957881	a@ad.ad	fd30fa5d31111e79bb15a0ff32484444614fec4acb831be4959d6a80f0ee875
a17a2790-8af2-4826-a396-657c5a00ebd9	kmat@mail.ru	63dbe0924bdfa9682cf76e59b80dff7389d95303d266620c59bf29f6523cb
eb26690f-4e10-4341-9435-1caa42ed6d20	marta@gmail.com	b667e27ed3c1de0b2568e696f96d52873f1cc218fb3ad9f13cceaf3d9117d5
a87f2efb-ec0d-4a57-b2fc-8c2e5ac401d3	wama@be.be	a2628ad7935d2a1977cf3f9996264e2a9dba4f2a169e585eb23570f5e07690
NULL	NULL	NULL

Installer « Let's Cook »

Google API

Pour utiliser les API de Google, il faut soit un téléphone Android officiel où une clé de licence développeur.

Obtenir une clé pour les map :

Génération du MD5

En premier, il faut récupérer le MD5 checksum de l'ID d'installation du SDK. Pour cela, il faut trouver le fichier *debug.keystore*. (Pour trouver son chemin, ouvrez les préférences d'Eclipse ; allez dans l'onglet Android, puis cliquez sur Build). Ensuite, il faut entrer cette ligne de commande en console :

```
$ keytool -list -alias androiddebugkey
        -keystore <chemin de la key>.keystore
        -storepass android -keypass android
```

Vous aurez votre MD5 checksum qui apparaitra.

Génération de la clé API

Il faut se rendre sur ce site : https://developers.google.com/maps/documentation/android/v1/maps-api-signup et insérer le MD5 checksum. De plus, il faut posséder un compte Google. La clé ainsi générée est unique et est utilisable uniquement pour l'ordinateur sur lequel vous avez généré le MD5[1].

Intégration au projet

Il faut se rendre sur le fichier xml suivant : maplayout.xml. Dans le fichier il faut se rendre à la ligne suivante :

```
android:apiKey="<votre-clé>"
```

Compilation

Les librairies nécessaires pour faire fonctionner l'application sont inclues dans le dossier « /libs » et ont été liées au moyen des chemins de compilation. Aucune modification n'est nécessaire à ce niveau-là.

Si jamais des soucis devaient malgré tout se présenter, il faut vérifier que la librairie MySQL et la librairie Android Binding sont bien dans le Build Path du projet.

[1] Source : https://developers.google.com/maps/documentation/android/v1/mapkey

Pour aller plus loin

Voici quelques idées que l'on peut appliquer pour rendre notre application meilleure encore :

Niveau fonctionnalité

- Implémenter un service web et un politique de sécurité cohérente
- Implémenter un système de notification pour les mails (serveur nécessaire)
- Ajouter aux utilisateurs un niveau « d'expertise », ainsi que des achievements.
- Pouvoir supprimer des objets si on a le niveau nécessaire uniquement (=protection contre le vandalisme)
- Avoir un historique personnel de nos actions et de celle des autres utilisateurs
- Pouvoir suivre des gens (comme sur Twitter)
- Quand on fait une recherche de supermarché pour une recette ou shopping list ou produits, si tous les produits ne sont pas trouvés proposer la combinaison des supermarchés les moins chères qui le permettrait (plutôt qu'une liste triée des marchés).
- Permettre une connexion via facebook,google,...
- Afficher la note moyenne d'un super market/recette
- …

Niveau code

- Ajouter plus de tests encore
- Corriger les défauts restants (s'il en reste)

Conclusion générale

Notre groupe pense vraiment que vu les fonctionnalités supportées et son architecture générale de bonne qualité (observables, chargement asynchrone et sure demande, système de cache) et ses quelques idées originales (messagerie interne, gestion des allergènes, connexion via email/SIM, …), on peut considérer qu'il se détache du lot, même si bien entendu certaines choses auraient sans doute pu être légèrement améliorées.

Vous avez vraiment des killer features ;
vous vous êtes bien cassé les pieds sur
l'architecture du modèle,
c'est pas mal du tout

Un ami programmeur